AD-A089 932 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2

A PROGRAM FOR THE CONVERSION OF PRODUCTIONS IN AN EXTENDED BACK--ETC(U)

JUL 80 E E MCCOY, T WETMORE

UNCLASSIFIED NPS52-80-010 NL

1 OF 1

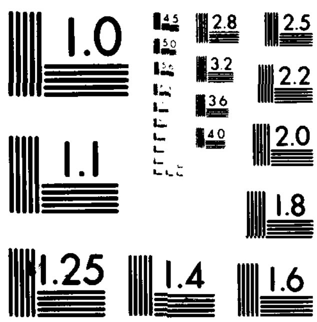
ADA 089932

END

DATE FILMED

11-80

DTIC



MICROCOPY RESOLUTION TEST CHART

NPS52-80-010

NAVAL POSTGRADUATE SCHOOL

Monterey, California

A PROGRAM FOR THE CONVERSION OF
PRODUCTIONS IN AN EXTENDED BACKUS-NAUR-FORM
TO AN EQUIVALENT BACKUS-NAUR-FORM

by

Earl E. McCoy
and
Thomas Wetmore III
University of Connecticut

July 1980

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

Jack R. Borsting
Provost

Reproduction of all or part of this research is authorized.

Earl E. McCoy

EARL E. McCOY
Visiting Assistant Professor of
Computer Science

Thomas Wetmore III

THOMAS WETMORE III
University of Connecticut

Reviewed by:

G. H. BRADLEY, Chairman
Department of Computer Science

Released by:

William M. Tolles

W. M. TOLLES
Dean of Research

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS-52-80-010 (14)	2. GOVT ACCESSION NO. AD-A089	3. RECIPIENT'S CATALOG NUMBER 933
4. TITLE (and Subtitle) (6) A PROGRAM FOR THE CONVERSION OF PRODUCTIONS IN AN EXTENDED BACKUS-NAUR-FORM TO AN EQUIVALENT BACKUS-NAUR-FORM.		5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report,
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Earl E. McCoy ~~and~~ Thomas Wetmore, III		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE (11) July 1980
		13. NUMBER OF PAGES (12) 18
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Conversion, Productions, Grammar, Backus-Naur-Form

(EBNF)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the use of a computer program that converts a grammar's production rules from extended Backus-Naur-Form to another equivalent set of production rules in ordinary Backus-Naur-Form suitable for use with the Yet Another Compiler-Compiler (YACC) system. This permits the language designer to use the far less bulky EBNF formats, and then to automatically convert to BNF for use with YACC. A PDP-11 computer system running the UNIX operating system is assumed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
(Page 1) S/N 0102-014-6601

Unclassified 251450

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT

This report describes the use of a computer program that converts a grammar's production rules from extended Backus-Naur-Form to another equivalent set of production rules in ordinary Backus-Naur-Form suitable for use with the Yet Another Compiler-Compiler (YACC) system. This permits the language designer to use the far less bulky EBNF formats, and then to automatically convert to BNF for use with YACC. A PDP-11 computer system running the UNIX operating system is assumed.

A

I Introduction

This report describes the use of a computer program that converts grammar production rules in an Extended Backus-Naur-Form (EBNF) into ordinary Backus-Naur-Form (BNF). EBNF is very convenient for a human description of a grammar but is not in a format acceptable to the Yet Another Compiler-Compiler (YACC) system [John75]. YACC requires the far more bulky format of ordinary BNF which is inconvenient for human use. The program whose use is described here is itself a translater written for the YACC system; the BNF it produces can be used for the input to YACC to yield a parse table and other processing for the original EBNF grammar.

The EBNF to BNF converter program is stored in the Naval Postgraduate School Computer Sciences Laboratory under the name "ebnftobnf". It is intended to work on a PDP-11 under the UNIX operating system. This technical report may be accessed on the UNIX system by typing "man ebnftobnf".

II The EBNF Syntax

The EBNF syntax acceptable as input to the converter is presented in this section. An example grammar is also presented.

EBNF makes use of grammar production rules consisting of terminals, nonterminals, and a replacement operator. In the discussion that follows we assume that terminal tokens are in uppercase letters or strings of letters or are enclosed in single quotes. The latter is usually reserved for trivial terminals such as parentheses, semicolons, etc. Nonterminals are lowercase letters or strings of letters. The head symbol is the nonterminal "z" as is the convention in some textbooks. The replacement operator is the left arrow, written as <--.

Two sets of metasymbols in EBNF must be removed from the grammar (by modifying the production rules) to produce an equivalent BNF grammar. These are the square brackets [...] meaning "zero or one", and curly brackets {...} meaning "zero or more". As is usual in production rules the vertical bar | means "or".

Consider the following example in EBNF:

    z <-- [A] C

In BNF two production rules are needed to express an equivalent grammar:

    z <-- C | A C

        or

    z <-- a' C

    a' <-- null | A

In either case the grammar accepts only the strings "C" or "AC".

Consider the use of the curly brackets to mean "zero or more":

    z <-- A {A}

This produces all the strings of the form A, AA, AAA, AAAA, anf so the BNF equivalent must be:

    z <-- z A | A

        or

    z <-- z A

z <-- A

The advantage of using EBNF to describe a grammar is obvious from these examples; it is unfortunate that YACC will not accept a grammar in this form. In the next section the exact format of the EBNF productions required for processing by YACC is presented.

III Use of the Converter Program

In this section a simple EBNF grammar is modified to the format acceptable to YACC, and the grammar converted to BNF by the translator program.

As an example grammer consider the following production rules:

z <-- {b} ;

b <-- [C] [A] D

Here z and b are nonterminals and A, C, D, and ; are terminals. How might these productions by modified to a format acceptable to the translator program?

Several symbols must be replaced in the EBNF used above to make productions acceptable to YACC. First, the replacement operator must be a colon (:) instead of a left arrow (<--). Secondly, all trivial terminals (ie. parentheses, semicolons, etc.) must be enclosed in single quotes ('). Thirdly, all other nonterminals must be explicitly indicated to YACC. Finally, the head symbol production rule must be the first (top) rule.

The above example production rules are manually converted to yield to following:

```
%token A C D
%%
z : {b} ';' ;
b : [C] [A] D ;
```

As many of the %token statements as needed can be used.

Now consider the execution of the EBNF to BNF translator. Since it is also a YACC program input it first must be executed:

```
yacc ebnftobnf
```

This produces a file in your file space named "y.tab.c.". The next step is to execute the C program in file "y.tab.c" by typing:

```
cc y.tab.c -ly
```

This produces a file named "a.out" that can actually translate EBNF to BNF by the following command:

```
a.out <ebnffile >bnffile
```

where "ebnffile" is the EBNF input file requiring translation; the ordinary BNF equivalent will result in file "bnffile". Choose whatever names you like for these files. The appendix shows the example presented above before and after translation.

IV Using the BNF Equivalent

In this section the use of the BNF equivalent as input to another YACC process is described.

The whole purpose of the EBNF to BNF conversion process was to produce a set of production rules acceptable to YACC, and thus be able to build a "compiler" that can process a "program" in the grammar to produce either a "yes" or "no" answer as to the program's syntax correctness or to compile it to some other target language. To accomplish this the equivalent BNF grammar must be embedded among other statements that indicate the terminal tokens and a C program (possibly making use of LEX [Lesk]).

To do this you must produce the same list of terminals used in the conversion process (%token ....., %%), and prepend it to the "bnffile". One VERY IMPORTANT production rule modification must be accomplished prior to resubmitting the "bnffile". The conversion process typically revises the order of the production rules due to the inclusion of new rules with new nonterminals. Be sure to insert the original head symbol production rule back at the very top of the list of rules; YACC requires this if a correct parse table is to result. It may have been moved down the list if it had square or curly brackets in its right hand side. Finally append any C program for processing the grammar into a target language to the list of production rules; separate them by a %% delimiter line. See the YACC manual for details.

VII Conclusion

This report describes how to convert a EBNF grammar to BNF suitable to YACC. While the program has been tested and found to work satisfactorily the usual disclaimer as to correctness must be made. The conversion process yields new production rules with new nonterminals. These new nonterminals are formed by concatenating the original nonterminals with prefixes such as "fst." and "opt.", and the results for a complicated grammar can get quite long. Use the editor to shorten them up if desired, but preserve the uniqueness of each nonterminal. Some nonterminals may contain sequences such as "._."; these are acceptable to YACC and so may be left unchanged.

BIBLIOGRAPHY

[John75] Johnson, Stephen C., "YACC - Yet Another Compiler-Compiler", Bell Laboratories, Murry Hill, NJ 07974.

[Lesk] Lesk, M. E., and E. Schmidt, "Lex - A Lexical Analyzer Generator", Bell Laboratories, Murry Hill, NJ 07974.

APPENDIX

```
**** The following is an example input file (ebnffile). ****


%token A C D
%%
z : {b} ';' ;
b : [C] [A] D ;


**** The following is an example output file (bnffile).  ****
**** Note that the first two rules must be interchanged  ****
**** if it is to be used as part of a YACC input via     ****
**** the a.out process.                                   ****
**** Note the null production: fst.b.:null|fst.b. b ;    ****


fst.b.:
        | fst.b. b  ;
z:
          fst.b. b ';'  ;
opt.C.:
        | C  ;
opt.A.:
        | A  ;
b:
          opt.C.  opt.A.  D   ;
```

**** Following is a listing of the "ebnftobnf" program ****

```
%token SYMBOL LITERAL
%{#define NULL 0
     struct node
     {  char symbol[30];
        struct node *first;
        struct node *next;
     };
     char symbol[30];
     struct node *pn;
%}
%%

grammar:
             rule_list;
rule_list:
          rule
        | rule_list rule;
rule:

                nonterm ':' alternative_list ';'
                  = {  printf ("%s%c0  ", $1->symbol, ':');
                       for (pn = $3; pn != NULL; pn = pn->next)
                    {  pitems (pn->first);
                          if (pn->next == NULL) printf (" ");
                       else printf ("0| ");
                          }
                            printf (";0);
                       }
nonterm:
                SYMBOL
                      = {  $$ = ncreate (symbol, NULL, NULL);
                        }
alternative_list:
                alternative
                       = {  $$ = ncreate ("a", $1, NULL);
                         }
                | alternative_list '|' alternative
                       = {  last ($1)->next = ncreate ("a", $3, NULL);
                         }
alternative:
                       = {  $$ = ncreate (" ", NULL, NULL);
                         }
                       | element_list;
element_list:
                element
                | element_list element
                       = {  last ($1)->next = $2;
                         }
element:
```

```
                    SYMBOL
                            = {  $$ = ncreate (symbol, NULL, NULL);
                              }
                    | LITERAL
                            = {  $$ = ncreate (symbol, NULL, NULL);
                              }
                    | '[' element_list ']'
                            = {  $$ = ncreate ("o", $2, NULL);
                                    if (!lookup ($$))
                                    {  printf ("0);
                                       pitem ($$);
                                       printf ("%c0 ", ':');
                                       pitems ($2);
                                       printf (";");
                                    }
                              }
             | '{' element_list '}'
                    = {  $$ = ncreate ("1", $2, NULL);
                            if (!lookup ($$))
                            {  printf ("0);
                               pitem ($$);
                               printf ("%c0 ", ':');
                               pitem ($$);
                               printf (" ");
                               pitems ($2);
                               printf (";");
                            }
                      }
%%
#define LETTER 'a'
#define DIGIT '0'

yylex ()
{
        int i, t, getch();
        char c;

    while ((c = getch()) == ' ' || c == '0 || c == '');
    if (type (c) == LETTER)
                {  i = 0;
           symbol[i++] = c;
                  while ((t = type (c = symbol[i++] = getch())) == LETTE
                       || t == DIGIT || c == '_' || c == '.');
                  ungetch(c);
                       symbol[--i] = ' ';
                  return (SYMBOL);
                }
            else if (c == ''')
                {  i = 0;
                       symbol[i++] = c;
                       while ((c = symbol[i++] = getch()) != ''');
                       symbol[i] = ' ';
                       return (LITERAL);
```

```
                }
                else return (c);
}
type (c)
char c;
{
        if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z') return (LETTER)
        if (c >= '0' && c <= '9') return (DIGIT);
        return (c);
}

ncreate (string, first, next)
char *string;
struct node *first, *next;
{
        struct node *p;

        p = alloc (40);
        strcpy (p->symbol, string);
        p->first = first;
        p->next = next;
        return (p);
}
last (np)
struct node *np;
{
        struct node *p;
        for (p = np; p->next != NULL; p = p->next);
        return (p);
}
strcpy (s, t)
char *s, *t;
{
        while (*s++ = *t++);
}
pitems (np)
struct node *np;
{
        struct node *p;

        for (p = np; p != NULL; p = p->next)
        { pitem (p);
          printf ("  ");
        }
}

pitem (np)
struct node *np;
{
        if (np->first == NULL) printf ("%s", np->symbol);
        else
        { if (strcmp (np->symbol, "o") == 0) printf ("opt");
          else printf ("fst");
```

```
            plist (np->first);
          }
}
plist (np)
struct node *np;
{
          while (np != NULL)
          {  if (np->first == NULL)
                  if (*(np->symbol) == ''') printf ("._");
                  else printf (".%s", np->symbol);
             else if (strcmp (np->symbol, "o") == 0)
             {  printf ("..opt");
                  plist (np->first);
                  }
                  else
                  { printf("..lst");
                   plist (np->first);
             }
         np = np->next;
     }
             printf (".");
}

strcmp (s, t)
char s[], t[];
{
          int i;
          i = 0;
          while (s[i] == t[i])
                  if (s[i++] == ' ') return (0);
          return (s[i] - t[i]);
}
char buf[1];
int bufp 0;
getch ()
{
          return ((bufp == 0) ? getchar() : buf[--bufp]);
}
ungetch (c)
int c;
{
          buf[bufp++] = c;
}

#define TRUE 1
#define FALSE 0
struct node *newnonterm [100];
int nonew 0;
lookup (np)
struct node *np;
{
          int i;
          for (i = 0; i < nonew; i++)
```

```
                if (equal (np, newnonterm[i]))
                  return (TRUE);
        newnonterm[nonew++] = np;
        return (FALSE);
}

equal (x, y)
struct node *x, *y;
{
        if (strcmp (x->symbol, y->symbol) != 0) return (FALSE);
        else return (eqlist (x->first, y->first));
}

eqlist (x, y)
struct node *x, *y;
{
        while (x != NULL && y != NULL)
        {  if (!eqtype (x, y)) return (FALSE);
           if (strcmp( x->symbol, y->symbol) != 0) return (FALSE);
           if (x->first != NULL)
                if (!eqlist (x->first, y->first)) return (FALSE);
           x = x->next;
           y = y->next;
        }
        if (x != y) return (FALSE);
        else return (TRUE);
}

eqtype (x, y)
struct node *x, *y;
{
        if (x->first == NULL) return (y->first == NULL);
        if (y->first == NULL) return (FALSE);
        if (*(x->symbol) == 'o') return (*(y->symbol) == 'o');
        return (*(y->symbol) == 'l');
}
```

INITIAL DISTRIBUTION LIST

1.	Earl E. McCoy Department of Computer Science Code 52My Naval Postgraduate School Monterey, California 93940	10
2.	Thomas Wetmore III Computer Science Department University of Connecticut Connecticut 06105	10
3.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
4.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
5.	Department of Computer Science Code 52 Naval Postgraduate School Monterey, California 93940	30